Fault-Tolerance for Matrix and Signal Processing Applications

Daniel S. Katz

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

Many applications are composed of a series of linear operations. In particular, the science data processing applications developed as part of NASA's Remote Exploration and Experimentation REE Project make heavy use of linear algebra and signal processing routines. The REE applications also are generally parallel programs, as multiple processors are needed to keep up with incoming data and respond to real-time events.

The goal of REE is to move ground-based supercomputing to space, and to enable new science to be performed on these supercomputers. REE is not willing to pay the power-performance penalty required when using radiation-hardened processors, and thus seeks to fly COTS components. However, running these processors in the galactic cosmic-ray environment found away from the Earth means that the processors will be subjected to transient faults. This paper discusses a number of the methods that have been developed, implemented, and tested by REE to allow these portions of these application to detect and possibly correct transient faults. These methods fall into the Result Checking subcategory of Algorithm-Based Fault Tolerance (ABFT, K. Huang and J.A. Abraham, "Algorithm–based fault tolerance for matrix operations," *IEEE Trans. on Computers*, 33(6):518–528, 1984, and P. Prata and J.G. Silva, "Algorithm based fault tolerance versus result-checking for matrix computations," *Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, pp. 4–11, 1999.) These methods can also be used to detect faults in many other situations, including any system with un-reliable hardware or network connections.

The routines that have been developed allow maintaining the same APIs as the standard versions of the routines. The fault-tolerance may either be hidden from the programmer, or exposed if greater control is desired. For REE, these APIs include BLAS (http://www.netlib.org/blas/), LAPACK (http://www.netlib.org/lapack/) , ScaLAPACK (http://www.netlib.org/scalapack/), PLAPACK (http://www.cs.utexas.edu/users/plapack/) , and FFTW (http://www.fftw.org/). While VSIPL (http://www.vsipl.org/) was not addressed by the libraries which have been implemented, this paper will discuss how this might be done (for general purpose processors), as well as providing examples of the current libraries, the current applications, and use of the libraries within the applications. The specific routines that have been protected include parallel forward and inverse FFTs, all single-processor BLAS3 operations, and parallel matrix-matrix multiplication, matrix inverse, LU decomposition, and SVD. Adding protection to the equivalent VSIPL routines would be almost trivial (assuming source is available, such as for the reference implementation or for a library developer), and adding protection to some other VSIPL routines, such as QR and Cholesky decompositions, would be fairly simple. The result checking routines take advantage of algorithms which perform a large number of operations on the data, and design checks which use far less operations. For example, a check on an FFT is O(n) while the FFT is O(n log n), and a check on a matrix-matrix multiply is $O(n^2)$ while the multiply is $O(n^3)$. Our

experience has been that these checks add overhead of 10-20% to the calculations. For this reason, VSIPL's scalar and vector operations cannot be protected by these means, as adding an O(n) check to a vector operation which is itself O(n) is not worthwhile, since it is simpler to just repeat the vector operation and see if the results are the same. Thus, the overhead for scalar and vector operations is approximately 100%. However, the vector operations are also faster than the matrix operations, and it has been out experience that they require a smaller fraction of the computer cycles, as least for the REE applications. This means that they are less likely to be affected by faults.

This paper also will discuss how one can take advantage of the memory hierarchy of modern systems in choosing how to implement fault detection. As shown in the automated ATLAS (http://www.netlib.org/atlas/) package, an optimized matrix-matrix multiply routine should be careful to first choose how to perform operations within the floating point and integer units on the processor, then how to use L1 cache, then L2 cache, etc. The tradeoffs of implementing fault detection at these levels will be discussed.